Ukraine Conflict Similar Tweets

Gabriele Cerizza

Università degli Studi di Milano
gabriele.cerizza@studenti.unimi.it
https://github.com/gabrielecerizza/amd_project

Introduction

In this report we detail our findings in the study of an algorithm capable of identifying similar pairs of documents within massive datasets. The experiments illustrated hereinafter were carried out as part of a project for the Algorithms for Massive Datasets course of Università degli Studi di Milano.

In Section 1 we illustrate the dataset and the adopted pre-processing techniques. In Section 2 we briefly describe the algorithm and its implementation. We also outline the neural network used as comparison. In Section 3 we comment our experiments. Finally, Section 4 contains our concluding remarks.

1 Dataset

In this section we provide an overview of the dataset (Section 1.1) and of the pre-processing techniques (Section 1.2).

1.1 Description

The dataset employed in our experiments was the "Ukraine Conflict Twitter Dataset" from Kaggle[1], released under the CC BY-NC-SA 4.0 license. We assume the reader to be familiar with the terminology associated with the Twitter platform, including expressions such as "tweet", "retweet", "hashtag", and "handle"[2].

This dataset boasts a total of over 40 million tweets concerning the conflict between Russia and Ukraine, which broke out on the 24th of February 2022. Those tweets were collected daily by monitoring hashtags. The dataset was first published on the 27th of February 2022. In the remainder of this report, we will refer to the version 127 of the dataset, downloaded on the 19th of June 2022.

The dataset comprises 109 compressed CSV files. For each sampled tweet, among others, we can find information concerning the author, the text, the hashtags, the language and the date of creation of the messages. We used the

[1] www.kaggle.com/datasets/bwandowando/ukraine-russian-crisis-twitter-dataset-1-2-m-rows
[2] We refer to https://help.twitter.com/en/resources/glossary for a quick review of the terminology.

`text` field to extract each document and the `language` field to select only English documents.

It is worth noting that the naming of the files is not consistent, which hinders attempts to process the tweets chronologically.

1.2 Filtering and Pre-processing

The dataset contains a sizeable number of retweets or tweets that differ only for the inclusion of a number or emoji or handle or punctuation symbol or URL. Tweets consisting only of hashtags or very short messages are likewise abundant. Given the nature of Twitter, this is to be expected. However, such documents do not provide a substantial challenge to models aimed at retrieving similar documents, given the amount of identical, overlapping text. For this reason, we set out to remove as much as possible those documents by way of a first filtering phase. The documents were then further pre-processed before being fed to the models.

Filtering. Filtering was carried out by performing the following operations:

1. we removed URLs;
2. we removed handles;
3. we replaced words with accents with their counterparts without accents;
4. we replaced special UNICODE characters with their ASCII counterparts, for instance "$\mathbb{R}$" with "R";
5. we replaced the ampersand with "and";
6. we converted everything to lowercase;
7. we removed non-ASCII characters, except for cyrillic characters;
8. we removed punctuation symbols;
9. we removed stop words;
10. we normalized the white space;
11. we dropped documents shorter than 100 characters.

These steps were performed having two objectives in mind: (i) dropping duplicates, after discarding noisy and irrelevant information; and (ii) reducing the number of different characters that could be found in the tweets. Indeed, a high number of different characters would force us to employ hash functions that adversely affect the execution time of the main algorithm (see Section 3.3).

Pre-processing. Different pre-processing pipelines were applied to the documents given as input to the main algorithm described in Section 2.1 and to the neural network described in Section 2.2.

Concerning the documents given as input to the main algorithm, the filtering steps described above also directly affected the documents and, therefore, may be considered part of the pre-processing pipeline. However, the filtering steps were also used to drop the documents that, after those operations, turned out to be identical. On the contrary, the pre-processing steps described below were not involved in determining whether two documents should be considered identical. These pre-processing steps were:

1. using the `spacy` library[3], we replaced each token with the corresponding lemma, to increase the match between words like "invasion" and "invaded";
2. we removed punctuation symbols that were introduced by the lemmatization;
3. we normalized the white space again.

The documents given as input to the neural network were not affected by the filtering steps, which, in this case, were used solely to drop duplicates. As a consequence, the only pre-processing steps were the following:

1. we removed URLs;
2. we replaced words with accents with their counterparts without accents;
3. we replaced the ampersand with "and";
4. we replaced special UNICODE characters with their ASCII counterparts, for instance "$\mathbb{R}$" with "R";
5. we removed non-ASCII characters, except for cyrillic characters;
6. we normalized the white space.

We decided to perform a light pre-processing for the neural network in order to leverage its ability to exploit the context of each word, which would have been hampered by removing punctuation and stop words or by lemmatizing the tokens.

Finally, note that, while numbers might be considered noise in some contexts, here we decided to keep them. Indeed, in our context, numbers could be found in dates and military equipment (e.g., the Russian T-72 tank or the M982 Excalibur 155 mm shell) and could help in discriminating the documents.

2 Models

In this section we briefly describe the algorithm used to find similar documents (Section 2.1) and the neural network we used as a baseline to compare performances (Section 2.2). We also propose an example to motivate the comparison between the two models (Section 2.3).

2.1 Locality-Sensitive Hashing Algorithm

In order to find the pairs of similar documents, we employed the algorithm described in [3], which can scale up to massive datasets. We assume the reader to be familiar with the algorithm and we provide here only a brief summary. The algorithm converts each document to a set of k-grams (shingles), builds a characteristic matrix and then a signature matrix by using hash functions, applies a locality-sensitive hashing (LSH) technique to find candidate similar pairs and, finally, checks the candidate pairs against the signature matrix to discard false positive pairs. Optionally, one could check the candidate pairs also against the characteristic matrix, if available.

[3] https://spacy.io/

Hash functions. Concerning the implementation details, we first discuss the various hash functions exploited by the algorithm. To avoid keeping the shingles in main memory, the algorithm hashes each shingle to a bucket, in the form of an integer. The total number of buckets associated to the hash function has a significant impact on the execution time. We generated different hash functions by truncating the output of the SHA-256 hash function at varying lengths of bits. In this way, we could experiment with different numbers of buckets.

The algorithm requires also hash functions that could map a row index to another row index. These hash functions are used to efficiently perform permutations of the rows. Since the number of these hash functions is a hyper-parameter set by the user, we needed a way to generate an arbitrary number of different hash functions. To this end, we followed the approach described in [4], generating hash functions of the form

$$h(x) = (ax + b) \bmod c \ ,$$

where x is a row number, a and b are random numbers smaller than the maximum row number and c is a prime number higher than the maximum row number.

Inside the LSH technique, to check whether two columns were equal, we hashed the string composed by the sequence of integers inside each column using the default `hash` function provided by Python. This allowed us to compress a potentially huge array of numbers into a single number, thus saving main memory space.

Characteristic matrix. In order to build the signature matrix, the algorithm also requires a characteristic matrix, where each column represents a document and each entry indicates whether or not a given shingle is contained in a given document. We store the characteristic matrix as a dictionary that maps each document index to its set of shingles. In this way, we do not waste memory to store the null entries.

LSH. Concerning the LSH technique, in order to find the number of bands for the signature matrix and the correlated number of rows in each band, we used `scipy` and solved the system of equations

$$\begin{cases} t = (\frac{1}{b})^{\frac{1}{r}} \\ b \cdot r = n \end{cases} , \tag{1}$$

where t is a threshold on the required similarity between two documents, b is the number of bands, r is the number of rows in each band and n is the total number of rows of the signature matrix.

Scalability bottlenecks. The implementation of the algorithm features two possible scalability bottlenecks. The first one concerns the characteristic matrix. Using 3-grams, 4-grams, 5-grams and 6-grams, each document contained on average 120-130 shingles. If we represented each shingle with a 32-bit integer, processing 10 million documents would require roughly 5 gigabytes of main memory ($\frac{1e7 \cdot 130 \cdot 32}{8e9} = 5.2$). Keeping the whole dataset (40 million documents) in main memory might prove challenging or even impossible. A solution could be to store the documents representations as sets of shingles in mass memory and read them one at a time when building the signature matrix. However, that would slow down an already time consuming operation, since we would have to cycle through the documents for each row of the characteristic matrix.

The second bottleneck concerns the number of similar pairs. The lower the value for the threshold, the higher the number of similar pairs identified by the algorithm. When processing millions of documents, even a moderate threshold might saturate the main memory. The only possible solution would be to store the candidate pairs identified by the LSH algorithm on mass memory. This would have to happen within each band of the signature matrix and, therefore, we would not be able to have unique pairs: the same pair might be identified and stored on mass memory in more than one band. This would result in redundant processing.

2.2 Neural Network (Transformer)

The dataset did not provide a ground truth against which we could compare our results. Therefore, we decided to use a state-of-the-art model in the field of natural language processing (NLP) to serve as a baseline in determining how similar two documents were. To this end, we opted to use MPNet [6], which is a neural network based on the revolutionary BERT Transformer [1, 7].

A Transformer used to obtain word embeddings is similar to word2vec [5], with one important difference. The difference is that word2vec produces static embeddings, which means that a given word will be mapped to the same vector regardless of its context. On the contrary, the Transformer produces contextualized embeddings, so that we will obtain a different vector for each different context (sentence) in which a given word appears. To give an example, the Transformer would generate different embeddings for the word “mouse” used with the meaning of animal and with the meaning of computer device, whereas word2vec would generate the same embedding.

We used the Transformer to obtain an embedding for each document. First, we extracted a word embedding for each word in a given document by using the last hidden layer of the model, which captures semantic features [2]. Afterwards, we computed the mean of the word embeddings of the document.

2.3 Motivation

We motivate our decision to compare the LSH algorithm with a Transformer by showing an example. Consider the following three documents.

1. “I went to the bank to withdraw money and compensate the plaintiff for their losses”.

2. “The Russians are withdrawing from the banks of the Dnipro River after suffering heavy losses”.
3. “After the judgement, I had to use my debit card and pay the suer for the damages”.

We can see that the first and the third document are semantically related, despite the fact that they do not share many words. On the contrary, the first and the second document are not semantically related, but share words like “bank”, “withdraw” and “losses”.

We apply a light pre-processing on the documents, consisting in removing punctuation, applying lowercase, performing lemmatization and normalizing the white space. Then, we extract 3-grams and we compute the Jaccard similarity between each pair of documents. Recall that the LSH algorithm produces an estimation of the Jaccard similarity. After that, we compute the cosine similarity between the sentence embeddings extracted from the Transformer, without any pre-processing.

The results are shown in Table 1. As expected, the Jaccard similarity identifies the first and second document as the most similar, whereas the Transformer correctly identifies the first and the third document as the most similar. This suggests that the LSH algorithm may not be able to adequately capture the semantic similarity between documents. Thus, we could use the Transformer to check whether the pairs identified as similar by the LSH algorithm are indeed similar.

Table 1. Similarity between each pair out of the three documents described in Section 2.3, computed by Jaccard similarity on 3-grams and by the Transformer.

	Similarity		
Method	Pair (1,2)	Pair (1,3)	Pair (2,3)
Jaccard Similarity	0.185	0.000	0.023
Transformer	0.192	0.648	0.131

3 Experiments

In this section, after reviewing some of the metrics used for evaluation (Section 3.1), we discuss our experiments on the LSH algorithm. These experiments concerned the growth of the number of different shingles and characters as we increased the number of documents (Section 3.2); the effect of the number of buckets on the hash functions used to map shingles to integers (Section 3.3); the effect of the threshold on the similarity between pairs (Section 3.4); the effect of the number of hash functions used to permute the rows of the characteristic matrix (Section 3.5); a final comparison between the LSH algorithm and the Transformer on 100,000 tweets (Section 3.6).

All the experiments were carried out using the files covering the tweets from the 1^{st} of April 2022 to the 7^{th} of April 2022. We selected only the tweets in the English language. We refer to Section 1.2 for the filtering and pre-processing techniques adopted. The analyzed files contained around 2 million documents. After the filtering, only 221,615 documents remained. This confirms that the dataset is filled with almost identical or very short tweets.

3.1 Metrics

We define the following metrics: precision, recall, F1-score and mean absolute error (MAE). Within a massive dataset, the number of pairs of dissimilar documents would be preponderant. Therefore, accuracy would likely be very high regardless of the capabilities of the algorithm. For this reason, we did not consider accuracy.

In this context, TP is the number of true positive pairs, FP is the number of false positive pairs, and FN is the number of false negative pairs. Those quantities were obtained by computing the actual Jaccard similarity between the documents, by using the characteristic matrix. Afterwards, we checked if the Jaccard similarity was higher than the threshold. If the algorithm identified a pair as similar, but the Jaccard similarity did not exceed the threshold, we considered the pair as false positive. If the algorithm identified a pair as similar and the Jaccard similarity exceeded the threshold, we considered the pair as true positive. If the algorithm identified a pair as not similar, but the Jaccard similarity exceeded the threshold, we considered the pair as false negative.

Precision is defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \,.$$

Recall is defined as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \,.$$

F1-score is defined as

$$\text{F1-score} = \frac{\text{TP}}{\text{TP} + \frac{1}{2}(\text{FP} + \text{FN})} \,.$$

MAE is defined as

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |y_i - x_i| \,,$$

where x and y are respectively the similarity estimated by the algorithm and the Jaccard similarity computed on the characteristic matrix. Recall that the output of the algorithm is a set of pairs of documents identified as similar. Thus, the algorithm does not provide a similarity value for all the pairs, but only for a subset. This being the case, N is not the total number of pairs, but the number of pairs identified as similar by the algorithm.

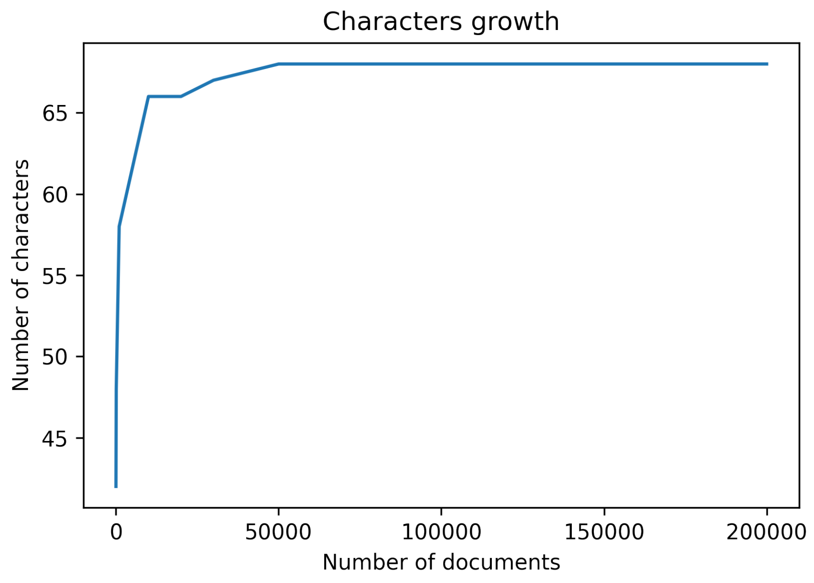

Fig. 1. Growth in the number of characters as the number of documents increases.

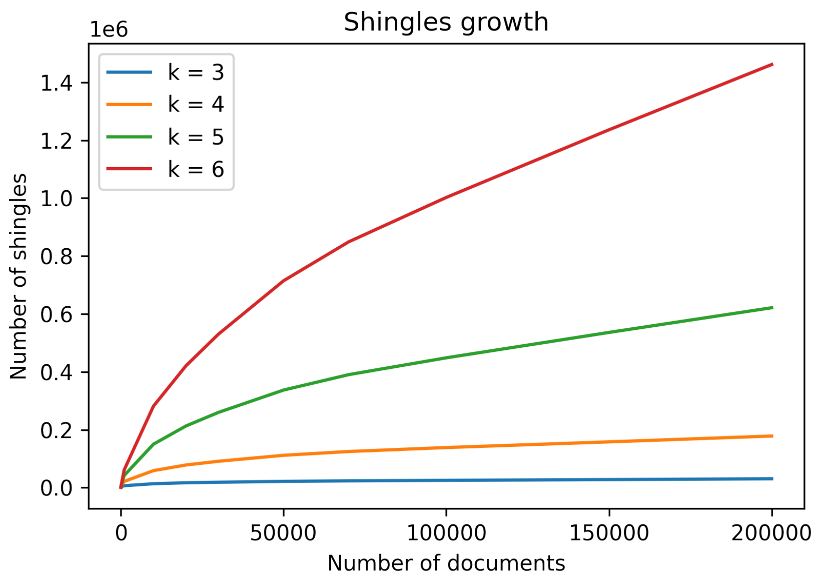

Fig. 2. Growth in the number of shingles as the number of documents increases. The growth is shown for shingles obtained with different k-grams.

3.2 Shingles and Characters Growth

In this first experiment we wanted to gauge how the number of different shingles (or k-grams) and characters within the whole corpus varied as the number of documents increased.

In Figure 1 we can see how the number of characters grew as the number of documents increased. In particular, we can see how, from 100,000 documents onwards, the number of characters was stable at around 65 characters (68, to be precise). This could be useful to get a hint on the appropriate k-gram with respect to the number of buckets of the hash function that maps k-grams to integers. For instance, given that $68^3 = 314,432$, we can infer that, if we used 3-grams, we would need a hash function with approximately 300,000 buckets.

Working with characters we can get an estimate on the number of buckets that we would need. However, it is unlikely that all the possible shingles will appear in practice. To this end, we also explored how the actual number of different shingles grew as the number of documents increased.

In Figure 2 we can see how the number of different shingles grew as the number of documents increased. It is not surprising to see that, with k-grams having a higher value of k, the number of shingles grew more steeply. We can also see, however, that this growth was approximately logarithmic in shape and stabilized quickly for low values of k. Indeed, for k equal to 3 and 4, most of the shingles had already been found at 50,000 documents. To be precise, at 50,000 documents we had 21,580 shingles with 3-grams and 111,643 shingles with 4-grams. Accordingly, for 3-grams, a hash function with buckets of at least 15 bits would be enough ($2^{15} = 32,768$ buckets). For 4-grams, we would need at least 17 bits to have a fair chance at avoiding collisions ($2^{17} = 131,072$ buckets).

3.3 Number of Buckets

In this experiment we wanted to see how increasing the number of bits, and thus the number of buckets, of the hash function that maps k-grams to integers affected various metrics. The experiment was carried out with a fixed threshold of 0.1, 100 hash functions for the permutations of the rows and a corpus of 100 documents.

The results are shown in Tables 2, 3 and 4 for 3-grams, 4-grams and 5-grams, respectively. We can formulate the following observations.

- Using a higher number of bits did not always yield a better result. For instance, we can see that with 3-grams 12 bits yielded the best F1-score. Although the corpus was limited to 100 documents, this might still suggest that collisions have only a mild impact on the performance.
- Across all the different numbers of bits, 3-grams obtained high values of F1-score most consistently.
- The higher the value of k in k-grams, the lower the number of pairs estimated as similar.

– Increasing the number of bits raised the execution time. With 22 bits, processing 100 documents took roughly 25 minutes. Coupled with a huge number of documents, this might make working with a high number of buckets prohibitive from the perspective of execution time.

Table 2. Effects of the increase in the number of bits for the hash function that maps k-grams to integers when k is equal to 3. We highlighted in bold the best results.

	Pairs No.	Recall	Precision	F1-Score	MAE	Time Delta
Hash Bits						
12	2166	0.775	**0.678**	**0.723**	0.025	0:00:04
14	2008	0.823	0.569	0.673	0.029	0:00:08
16	1761	0.815	0.579	0.677	**0.023**	0:00:24
18	995	0.525	0.634	0.574	**0.023**	0:01:29
19	1564	0.730	0.559	0.633	0.030	0:03:02
20	2108	**0.866**	0.492	0.628	0.033	0:05:53
22	1125	0.594	0.632	0.612	**0.023**	0:23:25

Table 3. Effects of the increase in the number of bits for the hash function that maps k-grams to integers when k is equal to 4. We highlighted in bold the best results.

	Pairs No.	Recall	Precision	F1-Score	MAE	Time Delta
Hash Bits						
12	404	0.656	0.317	0.427	0.030	0:00:03
14	187	0.598	0.326	0.422	0.030	0:00:07
16	237	0.820	0.308	0.448	0.031	0:00:24
18	245	0.782	0.278	0.410	0.032	0:01:32
19	353	**0.828**	0.204	0.327	0.039	0:03:06
20	348	0.690	0.172	0.276	0.039	0:06:37
22	191	0.747	**0.340**	**0.468**	**0.029**	0:23:55

3.4 Threshold

In this experiment we wanted to see how the value of the threshold affected the predictions. The experiment was carried out with a 16 bits hash function for the mapping between shingles and integers, 100 hash functions for the permutations of the rows and a corpus of 100 documents.

The results are shown in Tables 5, 6 and 7 for 3-grams, 4-grams and 5-grams, respectively. We can formulate the following observations.

Table 4. Effects of the increase in the number of bits for the hash function that maps k-grams to integers when k is equal to 5. We highlighted in bold the best results.

Hash Bits	Pairs No.	Recall	Precision	F1-Score	MAE	Time Delta
12	206	**0.750**	0.277	0.404	0.035	0:00:03
14	66	0.581	0.379	0.459	0.023	0:00:07
16	79	0.703	0.329	0.448	0.034	0:00:29
18	64	0.714	0.391	0.505	0.034	0:01:31
19	23	0.457	**0.696**	0.552	**0.022**	0:03:05
20	31	0.600	0.677	**0.636**	0.026	0:05:51
22	52	0.571	0.385	0.460	0.029	0:23:42

– Above a certain threshold, the algorithm reached a perfect F1-score for all the k-grams. In general, the algorithm behaved better when the threshold was either very low or very high. This is reasonable, since those are the easiest cases: when almost all pairs are similar or when only almost identical documents are similar.
– Increasing the threshold decreased the number of estimated similar pairs, as expected. However, this effect was stronger for k-grams with higher values of k.
– MAE got worse as the threshold increased. This could be attributed to the lower number of estimated similar pairs.

Table 5. Effects of the increase in the threshold value when working with 3-grams. We highlighted in bold the best results.

Threshold	Pairs No.	Recall	Precision	F1-Score	MAE
0.05	4067	0.924	0.941	0.932	**0.018**
0.10	1761	0.815	0.579	0.677	0.023
0.15	242	0.582	0.236	0.335	0.037
0.20	23	0.750	0.261	0.387	0.043
0.25	4	**1.000**	0.750	0.857	0.032
0.30	1	**1.000**	**1.000**	**1.000**	0.070
0.50	1	**1.000**	**1.000**	**1.000**	0.070

3.5 Number of Hash Functions

In this experiment we wanted to verify how the number of hash functions used to perform permutations affected the algorithm. The experiment was carried out

Table 6. Effects of the increase in the threshold value when working with 4-grams. We highlighted in bold the best results.

Threshold	Pairs No.	Recall	Precision	F1-Score	MAE
0.05	1819	0.791	0.650	0.714	**0.016**
0.10	237	0.820	0.308	0.448	0.031
0.15	25	0.875	0.280	0.424	0.050
0.20	8	**1.000**	0.375	0.545	0.058
0.25	2	**1.000**	0.500	0.667	0.071
0.30	1	**1.000**	**1.000**	**1.000**	0.108
0.50	1	**1.000**	**1.000**	**1.000**	0.108

Table 7. Effects of the increase in the threshold value when working with 5-grams. We highlighted in bold the best results.

Threshold	Pairs No.	Recall	Precision	F1-Score	MAE
0.05	1035	0.808	0.414	0.548	**0.019**
0.10	79	0.703	0.329	0.448	0.034
0.15	10	**1.000**	0.300	0.462	0.052
0.20	3	**1.000**	0.333	0.500	0.060
0.25	1	**1.000**	**1.000**	**1.000**	0.111
0.30	1	**1.000**	**1.000**	**1.000**	0.111
0.50	1	**1.000**	**1.000**	**1.000**	0.111

with a 16 bits hash function for the mapping between shingles and integers, a threshold value of 0.1 and a corpus of 100 documents.

The results are shown in Tables 8, 9 and 10 for 3-grams, 4-grams and 5-grams, respectively. We can formulate the following observations.

– Increasing the number of hash functions decreased the number of pairs estimated as similar and increased the precision. The effect is similar to what we observed in the threshold experiment, but with a significant difference. Here we can see how recall got progressively worse as we increased the number of hash functions. The takeaway is not to increase drastically the number of hash functions and not to rely on the sole precision to evaluate the algorithm. This will be important for our final experiment, because in that context we will be unable to measure recall and F1-score due to the huge number of documents.
– 200 hash functions seemed to strike a good compromise between precision and recall across all the different k-grams.
– The increase in execution time, when working with a higher number of hash functions, was negligible.

Table 8. Effects of the increase in the number of hash functions used to perform permutations when working with 3-grams. We highlighted in bold the best results.

	Pairs No.	Recall	Precision	F1-Score	MAE	Time Delta
Hash No.						
20	1793	0.577	0.403	0.474	0.045	0:00:19
100	1761	**0.815**	0.579	**0.677**	0.023	0:00:24
200	1468	0.725	0.619	0.668	0.023	0:00:31
300	74	0.046	0.770	0.086	**0.019**	0:00:36
500	1	0.001	**1.000**	0.002	0.032	0:00:45

Table 9. Effects of the increase in the number of hash functions used to perform permutations when working with 4-grams. We highlighted in bold the best results.

	Pairs No.	Recall	Precision	F1-Score	MAE	Time Delta
Hash No.						
20	951	0.798	0.075	0.137	0.057	0:00:21
100	237	**0.820**	0.308	0.448	0.031	0:00:25
200	137	0.775	0.504	**0.611**	0.020	0:00:34
300	17	0.169	**0.882**	0.283	**0.014**	0:00:40
500	0	0.000	0.000	0.000	0.000	0:00:45

Table 10. Effects of the increase in the number of hash functions used to perform permutations when working with 5-grams. We highlighted in bold the best results.

Hash No.	Pairs No.	Recall	Precision	F1-Score	MAE	Time Delta
20	772	**0.703**	0.034	0.064	0.068	0:00:19
100	79	**0.703**	0.329	0.448	0.034	0:00:25
200	36	0.514	0.528	**0.521**	**0.030**	0:00:29
300	3	0.081	**1.000**	0.150	0.043	0:00:33
500	0	0.000	0.000	0.000	0.000	0:00:43

3.6 Similarity between 100,000 Tweets

For this last experiment, we compared the results of the algorithm with different configurations of hyper-parameters against the Transformer. We used the Transformer to compute the similarity only between the pairs selected by the LSH algorithm. After that, we compared the rank-order correlation between the similarities computed by the two models using Kendall's Tau and Spearman's Rho. Both Kendall's Tau and Spearman's Rho range from -1 to +1, where -1 means strong disagreement, 0 means no correlation and +1 means strong agreement. We also considered Precision and MAE, as defined in Section 3.1. We forwent recall and F1-score due to the high number of documents.

The results are shown in Table 11. We can formulate the following observations.

- The strongest correlations were obtained by using 3-grams, with 4-grams being a close second.
- Surprisingly, even with 12-bits hash functions the algorithm achieved a good performance. This supports our hypothesis that collisions have a limited impact on the performance. Additionally, with 12-bits hash functions the algorithm is able to churn out results at a fast rate (around one hour for 100,000 documents).
- In general, the similarity values computed by the algorithm showed only a moderate positive correlation with the similarity values computed by the Transformer. Therefore, we can conclude that the pairs of documents estimated as similar by the algorithm may be lacking a semantic association.
- Increasing both the threshold and the number of hash function on 3-grams deteriorated the performance of the algorithm.

In Table we provide an example of similar documents identified by the algorithm having the best Kendall's Tau correlation in Table 11.

4 Conclusions

This work investigated the performance of the LSH algorithm described in Section 2.1 across different configurations of hyper-parameters. The algorithm

Table 11. Performance of the algorithm on a corpus of 100,000 documents. Kendall's Tau and Spearman's Rho correlations used the Transformer described in Section 2.2 to compare the similarity values. We highlighted in bold the best results.

K	T	Hash No.	Hash Bits	Pairs No.	Prec.	MAE	Kendall	Spearman
3	0.6	200	16	174941	0.956	0.023	0.229	0.323
	0.8	300	16	30917	0.903	0.015	0.076	0.109
	0.6	200	12	163221	0.967	0.020	0.230	0.319
5	0.6	200	16	94191	0.900	0.027	0.143	0.202

Doc. 1	Doc. 2	Similarity
#Russia's President Vladimir #Putin says he has signed a decree saying foreign buyers must pay in rubles for Russian gas from April 1, and contracts would be halted if these payments are not made.https://t.co/IUBuHMgw4n	#Putin - "Unfriendly countries" to start paying for Russian gas in rubles from April 1. Buyers not adhering to this will face gas contract termination by Russia.	0.728
'A lesson to the world #Ukraine #Germany, #Britain, #France They did not think of the people, but put idiots #Biden first and its economy last, and their persistence is the height of stupidity. Therefore, they should be replaced by people who put their people first	@bho Russia has ignored its obligations and worked to present	0.877

proved to be effective at processing massive amounts of data and at properly estimating the Jaccard similarity between documents. This is the field in which the algorithm excels. A neural network would not be able to compute the pairwise similarity between each pair of documents in a massive dataset. That would require an unreasonable amount of time.

The main pitfall of the algorithm is that Jaccard similarity on k-grams, as shown in Section 2.3, might not capture semantic similarity between documents. Given the fact that the algorithm is built on top of a Boolean characteristic matrix, this issue may prove difficult to address. The reason is that we cannot exploit state-of-the-art word embeddings, unless we find a way to suitably encode them as sets.

Another critical aspect of the algorithm was the execution time, at least with hash functions having a high number of buckets, as mentioned in Section 3.3. Indeed, building the signature matrix is a very expensive operation when using those hash functions. In [3] there are suggestions on how to address this issue, at the cost of a coarser estimation of the Jaccard similarity.

To further improve the execution time, we could also use a non-cryptographic hash function, instead of SHA-256, to generate hash functions with a variable number of buckets.

PUT THE BADGE IN README Example of similar tweets

References

1. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (Jun 2019). https://doi.org/10.18653/v1/N19-1423, https://aclanthology.org/N19-1423
2. Laicher, S., Kurtyigit, S., Schlechtweg, D., Kuhn, J., Schulte im Walde, S.: Explaining and improving BERT performance on lexical semantic change detection. In: Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop. pp. 192–202. Association for Computational Linguistics, Online (Apr 2021). https://doi.org/10.18653/v1/2021.eacl-srw.25, https://aclanthology.org/2021.eacl-srw.25
3. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, 3 edn. (2020). https://doi.org/10.1017/9781108684163
4. Liu, E.: Text Similarity. http://ethen8181.github.io/machine-learning/clustering_old/text_similarity/text_similarity.html (2015), [Online; accessed 23-June-2022]
5. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems. vol. 26. Curran Associates, Inc. (2013)
6. Song, K., Tan, X., Qin, T., Lu, J., Liu, T.Y.: Mpnet: Masked and permuted pre-training for language understanding (2020). https://doi.org/10.48550/ARXIV.2004.09297, https://arxiv.org/abs/2004.09297

7. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need (2017). https://doi.org/10.48550/ARXIV.1706.03762, https://arxiv.org/abs/1706.03762